VERDIFF: Vulnerability Presence Verification for Comprehensive Reporting Using Constraint Programming

Md Sakib Anwar
The Ohio State University
anwar.40@osu.edu

Carter Yagemann
The Ohio State University
yagemann.1@osu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

***Abstract*—Security practitioners often rely on a collaborative ecosystem of analysts and authorities to publicly disclose and track program vulnerabilities. Vital to these disclosures is the list of affected program versions, which stakeholders depend on to assess their security posture and plan appropriate responses. It is vital that these lists be *accurate* and *exhaustive* because 81.5% of industry systems rely on outdated dependencies and the average time to develop a patch is 256 days. Unfortunately, existing solutions for determining affected program versions do not scale to analyzing the entire release history. This paper presents VERDIFF, a framework that leverages a novel payload-guided, semantically enriched signature isomorphism matching specifically designed for swift, comprehensive vulnerability detection across *all* versions of a software program. Utilizing the initial vulnerable version found by an analyst and their crafted triggering input, VERDIFF formulates a distinct multi-level signature that is grounded in a strong correlation between dynamic binary analysis and source code signature matching, enabling a rapid high-level triage while accounting for nuanced low-level behaviors. Evaluating 27 CVEs spanning 11 programs, VERDIFF correctly pinpoints 265 misclassifications contained in official advisories.**

1. Introduction

In response to the growing prevalence of vulnerabilities, the cybersecurity community has developed a robust and standardized approach for identifying and disclosing their findings, centered around the Common Vulnerability Enumeration (CVE). This ecosystem relies on public vulnerability databases, such as the National Vulnerability Database (NVD), maintained by the National Institute of Science and Technology (NIST). Business security services [2], [5], [50], open source security tools [1], [3], [4], and security researchers [24], [25], [53] all rely on these records for the ground truth. Regrettably, recent studies [21], [44] have identified discrepancies in the affected versions of software reported for particular vulnerabilities. These inaccuracies lead to significant negative effects, including erroneous scientific assessments, incorrect evaluations of security standings, inappropriate incident responses, and additional problems.

The issue stems from maintainers prioritizing the *latest version of a program*, which compels analysts and reporters to do the same. Compiling and analyzing older program versions is impractical due to potential outdated dependencies, ranging from third-party libraries to specific kernel versions. An oversimplified solution to this problem is to treat all older versions as vulnerable and use the latest release; however, updating to a newer version of a dependency requires extensive refactoring, as 81.5% of systems [36] have outdated dependencies. In addition, many projects incorporate internal implementations of these third-party libraries, further complicating the matter. Moreover, the report [39] indicates that patch development takes an average of 256 days. Of greater concern is an earlier report [27] that reveals that 3 out of 30 vulnerabilities are exploited within hours of their disclosure, and 19 out of 30 within 12 weeks (84 days). Consequently, without a detailed list of affected versions in smaller libraries, vulnerabilities can remain undetected in larger projects.

Limitation of Existing Solutions. Identifying known vulnerabilities with a signature in a new domain, whether it be different programs or different versions of the same program, involves three crucial steps: identifying the vulnerable code, crafting the signature, and matching the signature. Based on the method used to detect the vulnerable code, existing work can be categorized into two primary groups: one utilizes predefined signatures or patterns and the other employs patches. Works using predefined signatures [15], [32], [48], [55], [67] focus on known vulnerability classes and can discover similar vulnerabilities in other programs. However, they lack comprehensive solutions for new vulnerability classes and cannot be directed to target specific vulnerabilities. A study [49] shows that different tools can detect different vulnerabilities in the same vulnerable code. Conversely, patch-based approaches [10], [35], [59], [60], [62], [64] can target specific vulnerabilities only after they have been acknowledged and fixed by developers. Moreover, they are limited by function granularity; they struggle with vulnerabilities across multiple functions or within function segments. We detail these limitations in §2.1.

Our Proposal. Based on our observations, we present VERDIFF, a payload-guided semantically enriched signature

isomorphism detection framework to perform an *automated* triage of *all* previously released versions of a program to construct a *comprehensive, timely, and accurate* list of vulnerable versions for a specific vulnerability. Unlike related tools VERDIFF considers the information available when a vulnerability is first discovered and provides a generalized solution that can be applied to a variety of vulnerability classes. We observe that for a wide class of vulnerabilities, the initial report often contains *steps to reproduce* along with expected vs. actual output. VERDIFF utilizes these Proof of Concept (PoC) triggering input to observe the manifestation of the vulnerability within a vulnerable program version to guide the creation of a graph-based vulnerability signature. VERDIFF carefully chooses which attributes of the program to keep or abstract in order to maintain a level of uniqueness that makes it distinguishable in a codebase, but comparable across all the other program versions. Additionally, it uses constraint programming to efficiently detect subgraph isomorphisms. Finally, it uses the signature to confirm the vulnerability in all other versions of the program. VERDIFF accomplishes this feat without the need for a patch or signature to identify the vulnerable code, and avoids manual compilation with sanitizers, thus sidestepping dependency and environmental issues.

Evaluation. We have implemented and evaluated our prototype on 27 CVEs spanning 10 CWEs for 11 well-known C/C++ programs (cpython, php, libpng) from the Linux Flaw Project [42], a well known dataset used for vulnerability detection evaluation [6], [57], [65]. Our prototype analyzes every version of our dataset in 1.82 hours (10.52 seconds per version, on average), discovering *265* misclassifications in official advisories. Of the 27 CVE reports, 17 CVEs (63%) have at least one misclassification in the latest updated official report. Among the related works mentioned we focus on publicly available works from each category, Tracer [32] (predefined signature-based detection) and Vuddy [35] (patch-based detection). VERDIFF demonstrates a remarkable improvement, achieving a 99.7% accuracy that surpasses the official report (62.01%), Tracer (55.36%) and Vuddy (61.80%). Furthermore, we carried out a real-world case study that examined all issues in an open source project over a period of one month, demonstrating the application of VERDIFF in the CI/CD pipeline.

2. Overview

This section presents the problem, explains current tool limitations, highlights obstacles and our insights, and describes our design assumptions.

2.1. Motivating Example

We examine CVE-2017-16353 in GraphicsMagick 1.3.26, a heap-based buffer overread vulnerability noted in Figure 1. The problem is in file `magick/describe.c` (Figure 1(c)), function `DescribeImage` on line 63, where variable `image` is user input. The

```
 61 define MaxTextExtent 2053
```

(a) magick/common.h

```
672 typedef struct _Image {
... ...
720 char *montage, *directory, *geometry;
... ...
879 } Image;
880
881 type = struct _ImageInfo  {
...   ...
961   char
962     magick[MaxTextExtent],
963     filename[MaxTextExtent];
...   ...
1001 } ImageInfo;
```

(b) magick/image.h

```
 63 MagickExport MagickPassFail
       DescribeImage(Image *image,FILE *file,
 64        const MagickBool verbose)
 65 {
...   ...
850   for (p=image->directory; *p != '\0'; p++)
851   {
852     q=p;
853       while ((*q != '\n') && (*q != '\0'))
854         q++;
855       (void) strncpy(image_info->filename,p,q-p);
...       ...
880   }
...   ...
907 }
```

(c) magick/describe.c

Figure 1: Vulnerable code for heap-based buffer over-read (CVE-2017-16353) of GraphicsMagick 1.3.26

value of `image->directory` is copied to `image_info->filename` on line 855 using function `strncpy`, with no size checks on fields `directory` or `filename`, risking heap overread. As indicated in file `magick/image.h` (Figure 1(b)), field `directory` has no defined size, while `filename` has size `MaxTextExtent`, decoded from file `magick/common.h` (Figure 1(a)) to be `2053` on line 61. The vulnerability is triggered by a crafted file making `image->directory` exceed `2053`. It was reported in October 2017, the CVE issued in November, and the fix in December. Our analysis shows all versions with the target file (`magick/describe.c`) from 1.3.7 (September 2009) to 1.3.26 (July 2017) are affected, but reports only identified 1.3.26. Let us consider the obstacles in identifying all affected versions using official report and related tools, Vuddy and Tracer.

Patch based Detection. It took 51 days to release the patch, including multiple commits, and without a public version control for GraphicsMagick, identifying the security patch commit is difficult. The CVE and release lack cross-references, complicating the issue. Despite knowing the vulnerable function, Vuddy struggles to pinpoint affected versions due to unrelated changes, accurately identifying only 2 out of 20 vulnerable versions, erroneously marking

18 as safe. A change causing Vuddy's failure is simply a change in format in a printed statement, not related to the execution or manifestation of the vulnerability.

Signature based Detection. Tracer has predefined signatures for many classes and operates at the instruction level. However, it lacks signatures for this specific CVE under CWEs "CWE-125: Out of bounds read" and "CWE-200: Exposure of Sensitive Information to an Unauthorized Actor", thus it cannot detect this vulnerability. Detecting it requires manual analysis to identify patterns. Notably, while analyzing GraphicsMagick version 1.3.26, Tracer found 171 API misuses leading to issues such as integer overflow and buffer overflow, but missed the targeted vulnerability.

2.2. Challenges & Insights

An ideal solution should identify vulnerable code at the instruction level, regardless of the types of vulnerability or the availability of patches, despite various challenges. When met, the solution can be seamlessly incorporated into the development pipeline of a project.

Identifying Vulnerable Code. Patch and predefined signature-based methods struggle to identify newer types of vulnerability without manual intervention. Interestingly, research [55] shows that many vulnerabilities depend on data flow, requiring manual identification of sources and sinks, limiting the breadth of the method. However, vulnerability reports often include a Proof of Concept (PoC) input that identifies and reaches the vulnerability directly. PoCs, essential in vulnerability [13], [30], [47] and exploitability assessments [14], [52], [58], are increasingly common due to fuzzing and symbolic execution [16], [19], [37], [68]. They allow data flow analysis without manually finding sources and sinks and accommodate non-standard patterns, such as "CWE-190 Integer Overflow or Wraparound."

Localizing Vulnerability. Despite the fact that these PoCs can trigger the vulnerability, the execution trace still includes a significant amount of superfluous code. For example, most PoCs are distributed as a file, which is initially read and converted into associated objects. This additional code can lead to an inaccurate signature and longer signature matching times. However, the extent of the vulnerability can be narrowed down or localized to a few relevant files using various process monitoring tools such as ASAN, MemCheck, etc. Utilizing these tools with the triggering input not only produces a stack trace to identify files of importance but also identifies the termination point for non-crashing vulnerabilities such as "CWE-20: Improper Input Validation".

Creating Signature. The triggering input creates a dynamic vulnerability signature, but compiling old program versions is tough due to outdated dependencies. Thus, signatures should be at the source code level, commonly available for OSS projects. The vulnerability model assists in identifying vulnerable code, but comparing source code alone is inadequate. Source code details, like a structure defined in a different file (`magick/image.h`) or field size determined elsewhere (`magick/common.h`), require resolution by the compiler. Therefore, source code must be abstracted with resolved values. When represented, the vulnerable code can be extracted, forming a signature.

Matching Signature. Hash matching is commonly used for signature comparison in research, with Vuddy creating function-level hashes for vulnerability detection. Our method focuses on instruction level, which requires prior knowledge of the vulnerable instruction, undermining matching efforts. Source code is often represented as a graph-like Abstract Syntax Tree (AST), with vulnerable code as a subgraph. As subgraph isomorphism is NP-complete, careful encoding of these representations is crucial. It's also important to maintain node positions so similar statements can be distinguished yet matched across versions.

2.3. Assumptions

We assume access to all versions of the open source code, with the ability to compile the vulnerable version and reproduce vulnerabilities using a PoC. The reporter or developer is expected to provide a data flow analysis or process monitoring log as per modern practices like ASAN reports. A version is considered vulnerable if the root vulnerable code is accessible, even if a PoC needs modification.

3. Design

This section explores insights from §2.2 and their application in our solution. The framework is divided into three parts: i) Identifying vulnerable code to model the vulnerability (§3.1), ii) Creating a static source code-based signature from the model (§3.2), and iii) Using the signature to detect the vulnerability in other versions (§3.3).

3.1. Modeling Vulnerability

Accurate identification of the code causing vulnerabilities is crucial in automated detection. The definition of vulnerable code varies by vulnerability type. In our motivating example, there are various errors contributing to the vulnerability, such as lacking validation when converting data to an object and not setting a maximum field size for `directory`. Lastly, the absence of validation for the resulting `q-p` affects data copied via the `strncpy` function. Conversely, examining a full PoC can lead to errors due to large signatures. Tools like ASAN and MemCheck help pinpoint the vulnerability's manifestation. If reports include matching filenames found with process monitoring, we use them to further localize the issue.

A summarized representation of the output of the data flow analysis using the PoC for our motivating example is illustrated in Figure 2. As observed, the execution undergoes memory allocation and traverses 22 files before reaching

```
MagickMalloc /.../magick/memory.c:156
MagickFree /.../magick/memory.c:509
AllocateString /.../magick/utility.c:216
...
713 lines across 22 files
...
DescribeImage /.../magick/describe.c:850
DescribeImage /.../magick/describe.c:853
DescribeImage /.../magick/describe.c:855
DescribeImage /.../magick/describe.c:856
DescribeImage /.../magick/describe.c:858
DescribeImage /.../magick/describe.c:860
ReadImage /.../magick/constitute.c:1451
ReadImage /.../magick/constitute.c:1472
ReadImage /.../magick/constitute.c:1473
```

Figure 2: Summarized data flow analysis output of motivating example CVE-2017-16353

```
Process terminating with default action of signal 6
          (SIGABRT): dumping core
at pthread_kill@@GLIBC_2.34(.../pthread_kill.c:44)
by raise (/.../posix/raise.c:26)
by abort (stdlib/./stdlib/abort.c:100)
...
by ReadImage (/.../magick/constitute.c:1473)
by DescribeImage (/.../magick/describe.c:860)
```

Figure 3: Crash stack trace report for motivating example CVE-2015-16353

our target file `describe.c`. However, we also notice that the data flow analysis does not stop where heap overread takes place (`describe.c:855`) since it does not lead to a crash. Interestingly, the program keeps running till line 860 where a call is made to the `ReadImage` function of the file `magick/constitute.c` and then faces the crash. That is, the crash is caused by reading the fields of `image_info` that have been overwritten due to the heap overread. This crash stack trace report is shown in detail in Figure 3 where the crash is said to originate from line 860 of `magick/describe.c`.

In contrast, Figure 4 presents an overview of the results of our process monitoring tool, which halts execution once a heap overread attempts to overwrite other fields. In other words, it can effectively detect the vulnerability at its origin rather than at the crash point. The process monitoring tool also specifies the exact location of the overread. This data is useful for defining the vulnerability's termination point during data flow analysis. Furthermore, the report refers to only 3 files, compared to the 26 identified in our comprehensive data flow analysis. However, since the CVE report pinpoints the vulnerability in `magick/describe.c` and matches our process monitor findings, we focus on that specific file.

Reflecting back to the data flow analysis in Figure 2, lines 850, 853, and 855 correspond to the pointer manipulation in Figure 1, leading to heap overread. Hence, a vulnerability model created from these lines will identify not only the vulnerability locus but also the relevant executed code. Furthermore, our localization approach cuts off the data flow at the root, creating an automated and specific *vulnerability model*.

```
Invalid write of size 1
at strncpy (/.../memcheck/vg_replace_strmem.c:604)
by DescribeImage (/.../magick/describe.c:855)
by IdentifyImageCommand (/.../magick/command.c:8394)
by MagickCommand (/.../magick/command.c:8869)
by GMCommandSingle (/.../magick/command.c:17396)
by GMCommand (/.../magick/command.c:17449)
by main (/.../utilities/gm.c:61)
```

Figure 4: Process monitoring output for motivating example CVE-2017-16353

3.2. Abstract Represenation for Signature Creation

Developing a vulnerability signature requires selecting a source code representation. We aimed to preserve semantics and balance uniqueness within a file and similarity across versions for nodes with similar syntax. Our approach uses attributes of each AST node and its children to create a *fingerprint* for each graph node. This method solves the issue of encoding multiple attributes into a concise form, enhancing the graph's generic, portable, and scalable nature. However, challenges arose in parsing source code, attribute selection, and efficiently transforming the representation, as discussed in the three components of fingerprinting.

AST Generation. The source code relies on abstractions like inline code, macros, and preprocessor directives, which a conventional AST does not address, as it focuses exclusively on lexical tokens. In addition, the major source code parsers [12], [22], [43] encounter issues with specific language features, such as `#pragma` directives and C++ support, requiring complete compiler-like emulation for a detailed analysis of the complexities of source code. Given these requirements, we choose `clang`, with a summarized output displayed in Figure 5. It efficiently parses function calls along with their signatures, determines the sizes and data types of data structure fields, and accounts for implicit casting, all of which are highlighted in the AST.

Encoding Attributes. The feature-rich AST generated by `clang` possesses extensive static source code characteristics. However, to leverage advanced graph algorithms, a concise encoding method is essential, representing source code as numbered nodes and edges. As illustrated in Figure 5, the nodes at the outset share lexical attributes and variable types but diverge in their operation type, exemplifying the need for property selection to preserve uniqueness. We have decided to include the following static properties of a program to define a node.

- **Node Kind.** The language construct abstraction of the syntax tree, i.e., the type of entity the current node is pointing to e.g., BinaryOperator, ImplicitCastExpr and CallExpr as seen in Figure 5.
- **Type Kind.** The type of node if applicable; for example, the type of variable (`filename` is `char[]` in Figure 1), return type of function, target type for binary operator, etc. The type often dictates the size, leading to different memory bugs, such as buffer

```
CStyleCastExpr <line:855> 'void' <ToVoid>
├─ CallExpr 'char *'
│  ├─ ImplicitCastExpr
│  │  'char *(*)(char *, const char *, unsigned long)'
│  │  └─ DeclRefExpr 'strncpy' (Function)
│  ├─ ImplicitCastExpr
│  │  'char *' <ArrayToPtr>
│  │  └─ MemberExpr
│  │     'filename' (char[2053])
│  │     └─ ImplicitCastExpr
│  │        'ImageInfo *' <LValueToRValue>
│  │        └─ DeclRefExpr 'image_info'
│  ├─ ImplicitCastExpr
│  │  'const char *' <NoOp>
│  │  └─ ImplicitCastExpr
│  │     'char *' <LValueToRValue>
│  │     └─ DeclRefExpr 'p'
│  └─ ImplicitCastExpr
│     'unsigned long' <IntCast>
│     └─ BinaryOperator '-' (q - p)
│        ├─ ImplicitCastExpr
│        │  'char *' <LValueToRValue>
│        │  └─ DeclRefExpr 'q'
│        └─ ImplicitCastExpr
│           'char *' <LValueToRValue>
│           └─ DeclRefExpr 'p'
```

Figure 5: AST as Generated by `clang` for `magick/describe.c:855` for CVE-2017-16353

overflow, integer overflow, and integer underflow as well.

- **Access Specifier.** Specifier when applicable, such as public, protected, or private. A change in a variable declaration that changes the access can lead to unexpected results by uncontrolled change to its value.
- **Concrete Values.** Concrete values associated with the nodes. For instance, for arrays the size (as shown in Figure 5), for binary operators and integer literals we consider the kind of binary operators, integer values, and so on.

Relative Positioning of AST Nodes. Hashing enriched nodes distinguishes syntactically different nodes but fails for similar ones, e.g., Binary AND operations share node kind, type kind, access specifier, and values. Specific properties such as line number must be avoided as they can change across versions. Integrating attributes from adjacent nodes distinguishes them by neighbors. The challenge is to determine the optimal number and selection criterion of neighbors, as common neighbors can cause incorrect matches. Inherited attributes result in similar attributes for nodes of the same rank with different children. To capture a node's execution variance, attributes must reflect subsequent instructions or observed children nodes, e.g. `case` statements under same `switch`. Limiting calculation layers reduces hash computation time and allows distant node changes to affect the graph, easing signature matching.

Subgraph Extraction for Signature Creation. Finally, we use the model from the last step to extract a subgraph from our abstract representation as a static signature for the vulnerability. While transforming the source code, we taint nodes in our model which are then extracted to create a subgraph, which may consist of multiple smaller trees, unlike our the initial file representing a single large tree. This is because data flow analysis can identify disconnected lines of code, such as in our example with lines 850, 853, and 855. Larger code sections can also create bigger signatures. By the end, the target files of all program versions are transformed into abstract representations, and a subgraph is extracted from the vulnerable version to act as our signature.

3.3. Subgraph Isomorphism for Vulnerability Detection

The final step involves utilizing the meticulously designed signature to compare it with all previous versions to identify the vulnerability. By pinpointing the vulnerable code and using process monitors and vulnerability reports to locate it, we refine the signature to be as accurate as possible. Additionally, by depicting the source code in an abstract graphical form and incorporating enriched static properties, we ensure the signature remains robust against syntactical changes that do not impact the vulnerability. Nevertheless, despite our thorough efforts, subgraph isomorphism continues to be an NP-complete problem, and for vulnerabilities with long signature, matching might demand a substantial amount of time.

Constraint Programming. Recent studies have demonstrated that constraint programming excels in subgraph isomorphism detection [7], [40], [51], [69], and various filtering approaches have been proposed to accelerate matching, consequently expediting vulnerability triage in our scenario. Constraint programming for subgraph isomorphism uses variables for potential vertex mappings. Constraints ensure valid mappings, resolved efficiently through inference and search, significantly reducing search space. We outline three such techniques and their correlation with source code and, ultimately, our abstract representation for quicker matching.

Terminology. To explain the techniques, we define our preliminary graph (abstract representation) derived from the vulnerable version as G_v (source graph), the vulnerability signature subgraph extracted from the initial graph as g (pattern graph), and the target graph generated from the previous version as G_t (target graph). The objective is to determine whether the pattern graph g is a subgraph of the target graph G_t.

Accelerating subgraph isomorphism requires each node in g to map uniquely to a node in G_t. *Injectivity* is achieved by including features from neighboring nodes, encoding the statement's relative position in its hash. Identical hashes may occur with duplicate code handling the same data in one file, though such duplicates are uncommon within a file [11], [28], [33], [34]. Another method from related work [69] filters target graph nodes based on *degrees*. A pattern graph node g can match only with a node in G_t that has an equal or greater outdegree. In source code, if a signature code

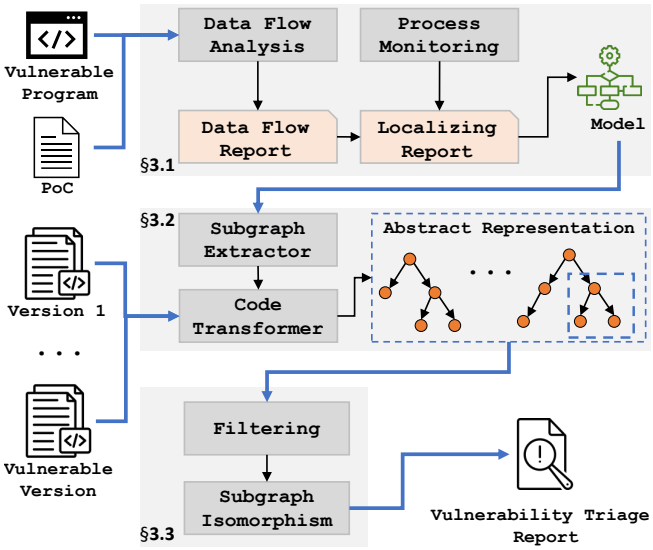

Figure 6: Overall design of VERDIFF

has a `switch` statement with 5 branches, it only matches a `switch` statement with at least 5 branches, aligning constructs with specific structural features. A final method filters by *path count and node distance* [8]. Code with equivalent path and distance configurations aligns logically with our signature. These filters significantly narrow the search space: injectivity and degree-based filtering restrict node matches, and path-distance alignment reflects structural similarities in the graph and code.

Partial Vulnerability Matching. Vulnerabilities that span multiple files and do not have an identified file designated in the disclosure matching our process monitoring report will exhibit a signature spread over various files. Signature for each file is separately matched for subgraph isomorphism. Consequently, subgraph isomorphism may succeed for some files and fail for others. In such scenarios, decisions must be based on the matched files. However, setting a global threshold (minimum percentage of files that must match to be deemed vulnerable) for different types of vulnerability across various programs with unique software development styles is challenging. Related work [9] highlights the importance of minimizing false negatives in high-stakes environments, and similarly, we aim to follow this approach. We achieve this by marking any version that matches any file of the signature.

If a signature spreads across a number of files, we can assume that each file performs some operation on the data. From our motivating example and the probable ways to solve it presented in §3.1 we realize that a fix is possible in every file involved. Thus, a target version matching only one file out of all the files in the signature might still cause the vulnerability to exist. Only when none of the files match our signature can we certainly say that the version is not vulnerable.

Overall Design. Figure 6 shows the overall design of VERDIFF which starts with the vulnerable program and a triggering input to create a vulnerability model with data flow analysis and process monitoring for localization. The second component takes the source code of all previous versions and transforms them into an abstract representation while identifying the signature in it. Lastly, the abstract representations and the signature are used to filter and finally run subgraph isomorphism in order to detect the presence of vulnerability in it.

4. Evaluation

Our evaluation results start with the experiment environment in §4.1, followed by the dataset evaluation in §4.2, and detailed results are provided in subsequent sections. We aimed to answer the following research questions with the evaluation of VERDIFF:

RQ1 To what extent can VERDIFF accurately confirm the presence of a vulnerability in all previous versions of a program?

RQ2 How does VERDIFF's vulnerability detection performance compare to that of other state-of-the-art tools?

RQ3 How accurate is the proof-of-concept (PoC) approach in identifying vulnerable code?

RQ4 To what extent is VERDIFF's methodology generalizable and effective for modeling and detecting vulnerabilities?

RQ5 How efficient is subgraph isomorphism with constraint programming in matching vulnerability signatures?

4.1. Experiment Environment

We implemented VERDIFF with clang-14.0.0 for AST generation and TaintGrind with Valgrind-3.20.0 for data flow analysis. We also used MemCheck and ASAN for process monitoring. Graphs were represented in LAD format, and we used the Glasgow Subgraph Solver [41] for efficient subgraph isomorphism. Docker version 24.0.5 and Ubuntu:22.04 were used as the base for our containers for sandboxing the evaluation. The programs were built with gcc, g++ version 11.3.0, cmake version 3.22.1, make version 4.3, automake version 1.16.5, and libtool version 2.4.6. Older program versions were built from the source as needed. The experiment was carried out on a single Intel Core i7-8700 3.2GHz machine with 16 GB DDR4-2666MHz memory running Linux Mint 21.

4.2. Dataset

For the evaluation, we chose the Linux Flaw Project [42], a dataset detailing C/C++ vulnerabilities in open source projects. Although NVD references various links and descriptions, it cannot isolate C/C++ vulnerabilities in open-source projects. Linux Flaw offers

CWE↓	Description	#CVEs
CWE-20	Improper Input Validation	9
CWE-119	Memory Buffer Misuse	80
CWE-125	Out-of-bounds Read	32
CWE-189	Numeric Errors	27
CWE-190	Integer Overflow or Wraparound	14
CWE-416	Use After Free	5
CWE-476	NULL Pointer Dereference	22
CWE-787	Out-of-bounds Write	14
CWE-noinfo/other	-	80

TABLE 1: Top CWEs with more than 5 CVEs in Linux Flaw Project

Category	Program	Used By
Graphics	autotrace	Inkscape, GIMP
	bento4	mprUI, Shaka Packager, ExoPlayer
	GraphicsMagick	Node.js, G'MIC, ImageOptim
	jasper	OpenCV, K Desktop, Copy.ai
	libpng	GIMP, Firefox, ClanLib, Chromium
	libtiff	GIMP, OpenCV, ImageMagick
	openjpeg	PHP, MySQL, KDE Ark
Compression	libzip	PHP, MySQL, KDE Ark
	zziplib	PHP, MySQL, SDL
Programming	PHP	Facebook, Wikipedia, MailChimp
	Python	Netflix, Spotify, YouTube, Instagram

TABLE 2: Evaluation Target Programs and Usages

detailed descriptions and root causes for recreating these vulnerabilities.

Common Weakness Enumeration (CWE). Table 1 lists leading CWEs with at least 5 CVEs from our dataset. Except for CWE-189, all are in the "2023 CWE Top 25 Most Dangerous Software Weaknesses," highlighting their severity. Most target memory safety issues like buffer overflow and out-of-bounds read/write, but also cover other vulnerability classes. Numeric errors (CWE-189) and integer overflow (CWE-190) aren't classified as memory safety problems but can cause them. Improper input validation (CWE-20) can lead to serious vulnerabilities like path traversal (CWE-22) or "CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component (Injection)," which includes SQL and code injection. We also include CVEs with insufficient information (CWE-noinfo) or those uncategorized (CWE-other). Our dataset reflects the scope and reality of real-world vulnerabilities, focusing on critical security risks.

Programs. Previous research on code reuse and vulnerability spread focuses on large projects integrating smaller open source ones. Conversely, our scenario emphasizes the programs impacting downstream users. The study [54] reveals 40% of libraries are dependencies for 84% of software, especially in compression, programming, and graphics. Acknowledging their downstream importance, we choose programs affecting end-user products detailed in Table 2. These include media parsing libraries like bento4 and libpng, used in applications like Firefox and Chromium. Tools such as libzip and zziplib support interpreters like PHP and databases like MySQL. We also analyze vulnerabilities in scripting interpreters like PHP and Python with C back-ends, demonstrating the flexibility of our solution across different program complexities. Our selected programs' dependencies show the impact of open-source vulnerabilities; for instance, a flaw in libpng may compromise Chrome or Firefox. Similarly, vulnerabilities in the analyzed interpreters pose risks to various applications, users, and developers. Our dataset answers our **RQ4**, demonstrating the general applicability of VERDIFF.

4.3. Ground Truth

In order to establish the ground truth, we define both *vulnerable* and *safe* versions of a program. A program might contain multiple vulnerabilities, each with different proofs of concept (PoCs) that activate them. Multiple methods might trigger a particular vulnerability. The vulnerable version is defined where the core vulnerable code is present. A PoC for a new release may not work for an older version, but the vulnerability can still exist. A PoC might crash before reaching the intended vulnerability in older versions. We address these scenarios in defining the ground truth. A version is marked safe if the vulnerable code is absent, meaning that the PoC or any variation cannot trigger the vulnerability. Thus, the primary bottleneck in our evaluation has been establishing the ground truth for each version of the program in relation to each vulnerability. Analyzing a vulnerability manually required understanding its root cause and the mechanism by which the PoC exploits it. To gain a better understanding of vulnerabilities, we used ASAN, UBSAN, MemCheck, perf, and gdb as needed. Our ground truths were determined through a three-layered approach.

1) **Executing PoC on Target Version.** First, we compile and run the PoC against the target version. Older program versions required specific environments. For example, php-5.0.4 needs libxml, zlib, libzip, zip, m4, flex, autoconf-2.59, re2c-0.13.4, bison-2.4, and libtool-1.4.3. Setting up these environments often involves building several dependencies from scratch. We have utilized docker to create a sandbox and ensure that it is the only version installed for testing.
2) **Sanitizers.** Second, we compile the versions again with sanitizers such as ASAN and run the PoC. For vulnerabilities that do not cause crashes, sanitizers provide evidence of the vulnerability. Even with vulnerabilities that do cause crashes, we use sanitizers to confirm that the crash is due to the specified vulnerability. That is, although sometimes the PoC may cause a crash, the reason might differ in different versions.

						Abstract Repr.		Signature		Time (s)				Official Report		Tracer		Vuddy		VerDiff w/o local.		VerDiff	
CWE	**Prog.**	**CVE**	**CVSS**	**#Ver.**	**#Vuln**	**#Nodes**	**#Edges**	**#Node**	**#Edges**	**Gen.**	**Match**	**Total**	**/Ver.**	**#FP**	**#FN**	**#FP**	**#FN**	**#FP**	**#FN**	**#FP**	**#FN**	**#FP**	**#FN**
	libtiff	2017-7599	7.8	13	13	122,723	125,774	55,115	57,950	665.34	64.04	731.07	56.24	0	12	0	13	-	-	0	0	0	0
CWE-20	**libtiff**	2017-7600	7.8	13	13	186,105	189,558	59,966	63,016	732.59	65.81	803.22	61.79	0	12	0	13	-	-	0	0	0	0
	libtiff	2017-7601	7.8	13	13	5,220,881	5,529,728	51,569	54,260	745.07	69.22	820.88	63.14	0	12	0	13	-	-	0	0	0	0
	bento4	2017-14261	7.8	27	27	17,395	18,375	497	525	0.51	2.45	2.96	0.11	0	26	-	-	-	-	0	0	0	0
CWE-119	**libtiff**	2016-10092	7.8	19	19	11,822	12,811	514	557	0.63	0.07	1.07	0.06	0	0	-	-	*	*	0	0	0	0
	autotrace	2017-9151	9.8	14	10	26,404	27,216	1,886	1,944	7.93	0.69	8.97	0.64	0	9	0	10	0	0	0	0	0	0
	zziplib	2017-5978	5.5	13	13	25,744	26,912	1,609	1,682	0.38	5.06	5.44	0.42	0	12	0	13	0	0	0	0	0	0
CWE-125	**GM**	2017-12937	8.8	53	6	1,793	1,732	1,037	1,039	7.86	0.69	8.73	0.16	0	5	0	6	0	1	47	0	0	0
	GM	2017-16353	6.5	53	20	139,686	139,903	4,506	4,513	2.56	42.72	45.29	0.85	0	19	0	20	0	18	33	0	0	0
	PHP	2013-7226	6.8	52	9	871,468	877,032	16,759	16,866	1,465.01	141.22	1,610.33	30.97	0	0	0	9	*	*	43	0	0	0
CWE-189	**PHP**	2012-2386	7.5	27	13	53,788	54,236	3,842	3,874	20.46	1.96	22.77	0.84	14	0	0	13	-	-	14	0	0	0
	PHP	2010-1866	9.8	16	3	386,342	394,534	16,450	16,706	94.93	8.39	109.12	6.82	0	0	0	3	-	-	0	0	0	0
	autotrace	2017-9162	9.8	14	11	169,062	174,228	6,866	7,044	21.68	1.42	24.92	1.78	0	10	-	-	-	-	0	0	0	0
CWE-190	**jasper**	2016-10251	7.8	19	19	32,737	33,706	1,723	1,774	7.67	0.55	8.33	0.44	0	0	0	19	0	1	0	0	0	0
	python	2016-5636	9.8	28	28	505,911	512,847	9,057	9,189	68.09	5.83	77.11	2.75	0	0	0	28	0	23	0	0	0	0
	libpng	2019-7317	5.3	36	36	273,672	280,512	7,602	7,792	21.10	808.64	829.74	23.05	0	0	-	-	0	0	0	0	0	0
CWE-416	**libzip**	2019-17582	9.8	19	1	54,320	55,440	2,716	2,772	1.15	11.96	13.11	0.69	0	0	-	-	0	0	18	0	0	0
	autotrace	2017-9182	7.5	14	11	611,739	622,812	15,753	16,065	108.33	9.73	121.06	8.65	0	10	-	-	0	7	0	0	0	1
	bento4	2017-14640	6.5	28	28	35,460	37,296	985	1,036	0.51	4.96	5.48	0.20	0	27	-	-	0	26	0	0	0	0
CWE-476	**openjpeg**	2016-7445	7.5	12	1	120,088	120,751	7,064	7,103	5.12	113.90	119.02	9.92	0	0	-	-	-	-	0	0	0	0
	zziplib	2017-5980	5.5	13	13	25,744	26,912	1,609	1,682	0.38	4.89	5.27	0.41	0	12	0	13	0	0	0	1	0	1
	bento4	2017-14644	8.8	28	28	16,272	16,956	452	471	0.08	2.21	2.29	0.08	0	27	-	-	-	-	0	0	0	0
CWE-787	**jasper**	2016-9560	7.8	25	25	40,920	41,340	1,364	1,378	0.26	6.80	7.06	0.28	0	0	0	25	0	0	0	0	0	0
	zziplib	2017-5975	5.5	13	13	12,288	12,240	256	255	0.56	0.66	1.22	0.09	0	6	0	13	0	0	0	0	0	0
Noinfo/ other	**jasper**	2017-5502	5.5	17	17	142,433	144,629	1,294	1,317	0.26	5.35	5.61	0.33	0	16	0	17	0	0	0	0	0	0
	libpng	2004-0597	N/A	91	55	1,075,080	1,110,984	10,540	10,892	11.30	1,122.57	1,133.88	12.46	36	0	0	55	-	-	36	0	0	0
	PHP	2020-7066	4.3	29	29	44,973	45,030	789	790	24.86	0.27	28.85	0.99	0	0	0	29	-	-	0	0	0	0
Total				699	474	10,224,850	10,633,494	281,820	292,492	4,014.62	2,502.08	6,552.77	284.16	**50**	**215**	**0**	**312**	**0**	**76**	**191**	**1**	**0**	**2**
Average				25.89	17.56	378,698	393,833	10,438	10,833	148.69	92.67	242.70	10.52										

TABLE 3: Detailed Evaluation of VerDiff

3) **Manual Code Verification.** Finally, we analyze the vulnerability, the execution trace, stack trace, and core dump if necessary to identify the root cause of the vulnerability. We verify each version to ensure consistent results from PoC executions and sanitizers. For example, CVE-2017-7599 in libtiff is not triggered by the PoC for version 3.9.3 due to the lack of `bigtiff` image support. That is both PoC execution and execution with sanitizers fail to prove the existence of the vulnerability. However, manual examination shows that the vulnerability persists for other image types, i.e. a modified PoC can trigger the vulnerability.

4.4. Result Overview

Table 3 provides an assessment of VerDiff, starting with *metadata* for each vulnerability, the CWE linked to CVEs, the affected software, and the CVE itself. It includes the CVSS Score, the number of versions tested, and vulnerable versions identified via ground truth analysis, indicating the severity and impact of the vulnerability. The CVSS score is based on severity and exploitability, considering exploitation vectors and privileges. The table shows the size of the Abstract Representation and Signature and the time to generate and match the signature with statistics. It also includes the official report, results from a comparative study with Tracer and Vuddy, and outcomes for VerDiff, with and without localization.

As can be seen, in total VerDiff analyzes 699 versions of the 11 programs targeted and 25.89 versions on average for each vulnerability. The total size of the abstract representation is 10,224,850 nodes and 10,633,494 edges, which comes down to 378,698 nodes and 393,822 edges on average. On the other hand, the signature contains on average 10,438 nodes and 10,833 edges, which is only 2.7% of the abstract representation.

On average, it takes 148.69 seconds to transform all the files associated with a vulnerability signature into an abstract representation. However, it takes 92.67 seconds on average to match the generated signature with all previous versions of a program. On average, it takes VerDiff 242.70 seconds to transform all the necessary code and match it with the signature. Lastly, it takes on average only 10.52 seconds to transform the source code and match it for vulnerability detection. Our analysis also shows that there are 265 mismatches in official reports, that is, versions that are incorrectly identified. On the other hand, VerDiff only has 2 such cases in the entire dataset. *Overall, the official reports from the CNAs contain 215 FNs and 50 FPs, versus 2 FN and 0 FPs for* VerDiff.

4.5. Result Interpretation

Metadata. For each CWE, we have tried to diversify the programs and choose programs that have more severe downstream impact. To verify our assumptions, we have used a related work [54] that studies third-party library dependency in C/C++ ecosystem. We use their dataset to study the number of dependencies for each program under a CWE and choose programs that exihibit the most dependencies. This lets us assess the risk of misinformation regarding vulnerabilities in these programs. For some cases, we have chosen to focus on a particular program, since other programs under that CWE did not show much impact as a dependency. This also shows how VERDIFF is able to distinguish similar vulnerabilities targeting different parts of a codebase.

The table then shows the CVSS score that ranges from 4.3 to 9.8 and shows the range of CVEs that can be modeled and analyzed with VERDIFF. The number of versions analyzed depends on the number of releases prior to the version in which the vulnerability is first discovered. On the other hand, the number of vulnerable versions is determined using our three-layer ground-truth establishing method explained in appendix §4.3. We excluded versions where we could not establish the ground truth due to compilation issue and ambiguous root cause.

Modeling & Slicing. The table presents the node and edge counts, indicating graph size, for both the abstract representation and its derived signature. The graph mapping the source code ranges from 1,793 to 5,220,881 nodes and 1,732 to 5,529,728 edges. Within a given CWE and program, such as libtiff under CWE-20, varying graph sizes may result from different vulnerabilities traversing multiple files. On the other hand, the signatures contain between 256 and 59,966 nodes and between 255 and 63,016 edges. The size of the signature reflects the actual execution of the instructions that lead to the vulnerability. This stark difference emphasizes how minimal code portions contribute to vulnerabilities, highlighting the need for precise identification of vulnerable code. Our localization approach helps in this precise identification of vulnerable code.

Time. Subsequently, the table presents the time required for VERDIFF to create the abstract representation and signature, followed by the time required to match this signature with other versions. On average, VERDIFF takes 10.52 seconds to produce the abstract representation, generate the signature, and verify its presence by matching the signature for an individual version. It is also apparent that a considerable amount of time is dedicated to abstraction of the code and then constructing the signature. However, the abstract representation of the source code files can be cached for a faster analysis by VERDIFF. For verifying all 699 versions in relation to the 27 targeted CVEs, VERDIFF takes approximately 2,502.08 seconds, which is equivalent to approximately 3.6 seconds per version, which underscores the efficient facilitation of subgraph isomorphism provided by constraint programming within VERDIFF. This also answers our **RQ5** showing the efficient and swift signature matching of VERDIFF using constraint programming based subgraph isomorphism.

Comparative Analysis. We have compared VERDIFF with two state-of-the-art tools namely Tracer and Vuddy and conducted an ablation to study the effect of our understanding into the problem. The comparative analysis in Table 3 first shows the official report that was collected from the National Vulnerability Database (NVD) maintained by NIST. We chose to compare VERDIFF with the current version of the report to show the present security risks. That is, although when the vulnerabilities were first discovered, there might be more misclassifications of affected versions, we chose to work on the most updated version of these reports. Next, the table shows the result from Tracer and Vuddy which uses predefined signatures and patch-based function hashes to verify the range of the vulnerability. This answers our **RQ2** showing why contemporary tools cannot offer a general and practical solution to the problem.

VERDIFF **vs Official Report.** We compare VERDIFF with the current reported versions and show the discrepancy between our finding and the official reports. As shown, most vulnerability reports contain on average 9.30 false negatives where a vulnerable version is labeled nonvulnerable, which has a serious impact. The reasoning behind this has been explained in appendix §5.2. Of the 27 CVEs we have analyzed, only 10 have accurate information regarding the affected list of versions. It can be noticed that the accuracy of VERDIFF is neither dependent on the size of the signature nor the time it takes to match, that is, we are able to capture and match for complex types of vulnerabilities. Furthermore, we have compared VERDIFF with two state-of-the-art tools that employ static signature-based vulnerability, namely Vuddy [35] and Tracer [32].

VERDIFF **vs Tracer.** Tracer uses predefined vulnerability signatures to detect recurring vulnerabilities, compiling a dataset across several CVEs and OSS. Its main goal is to identify new vulnerabilities seen in other programs. However, Tracer fails to detect targeted vulnerabilities from the dataset. While it effectively finds new memory leaks and ranks memory-related API misuse, it is not suitable for targeted detection. Tracer relies on Infer by Meta, which requires the target program's build system to follow the "configure, make, make install" approach and fails with cmake. Our efforts to create a suitable environment resulted in failures marked with (-) in Table 3. Consequently, Tracer performs worse than expected, as it was not meant for certain scenarios but for detecting recurring vulnerabilities using its signature dataset. Details of the evaluation environment are in appendix §B.1.

VERDIFF **vs Vuddy.** Vuddy detects vulnerable code clones by fingerprinting functions and comparing them to known vulnerable fingerprints. Since we operate under the assumption of no patch available, we have relied on CVE or git issue reports for identifying function for Vuddy. If function names are unknown, they are marked with (-) in Table 3.

When possible, Vuddy creates a vulnerable function signature to match with other versions. Details on Vuddy's usage are in appendix §B.2. Despite its speed, Vuddy struggles with identifying affected versions as it focuses on single functions. It misses vulnerabilities beyond functions or files and can be misled by unrelated function changes, as shown in §2.1. Incorrect function names in reports marked with (*) also hinder Vuddy's performance, leading to exclusion from results. Overall, Vuddy effectively detects vulnerabilities in 14 of 27 CVEs but still produces many false negatives.

Ablation Study. We conducted an ablation study to assess the effectiveness of VERDIFF's localization technique. For vulnerabilities affecting *all previous versions*, the study yielded consistent results as our localized files harmonize with the overall signature. In other cases, localization might expand the signature by including irrelevant code. Without localization, the signature overestimates vulnerabilities, matching more with older versions, as VERDIFF shows by producing 191 false positives compared to none with localization. Localization significantly impacts CWE-125 (Out-of-bounds read) and CWE-189 (Numeric errors). Our example in §2.1 illustrates CWE-125, where continued program execution after an over-read complicates data flow analysis. Similarly, numeric errors reduce signature precision since they don't halt execution. **RQ3** is addressed by recognizing that PoC often contains extra code, causing false positives. Using sanitizers refines the model to improve precision.

FP-FN Analysis. A false positive occurs when a version is wrongly identified as vulnerable, while a false negative occurs when a vulnerable version is not recognized. Our analysis shows 251 false negatives in official reports, where vulnerable versions are not included. There is also a case with 14 false positives, where versions are incorrectly labeled vulnerable. Often, only one version is marked as vulnerable when all prior versions are equally at risk. Official documents may list affected version ranges, but some compromised versions are missing. For example, under CWE-787 for CVE-2017-5975 in zziplib, all versions are vulnerable, but only seven are reported. For CVE-2004-0597 in libpng, documentation lists 19 of 55 affected versions.

We examine certain CVEs with false positives and negatives for VERDIFF, explaining why they occur. VERDIFF has no false positives but two false negatives due to our design choices in defining and narrowing vulnerable code. These negatives occur in CVE-2017-5980 of zziplib and CVE-2017-9182 of autotrace because our signature matching fails, yet vulnerabilities remain. This happened when a vulnerability remained despite changes in a function call signature, such as added arguments, that altered the execution of the program. It is challenging to assess whether these changes affect vulnerability. Our **RQ1** confirms a 99.7% accuracy, surpassing the compared tools. Details of how ground truth was established are mentioned in Appendix §4.3.

Program		Signature				
Name	**Size (MB)**	**#Nodes**	**#Edges**	**Gen.**	**Match**	**Total**
autotrace	8.81	8,168.33	8,351.00	45.98	3.95	3.69
bento4	17.00	644.67	677.33	0.37	3.21	0.13
GM	38.00	2,771.50	2,776.00	5.21	21.70	0.51
jasper	5.00	1,460.33	1,489.67	2.73	4.23	0.35
libpng	3.40	9,071.00	9,342.00	16.20	965.61	17.75
libtiff	11.00	41,791.00	43,945.75	535.91	49.78	45.31
libzip	8.20	2,716.00	2,772.00	1.15	11.96	0.69
openjpeg	12.00	7,064.00	7,103.00	5.12	113.90	9.92
PHP	162.00	9,460.00	9,559.00	401.31	37.96	9.91
python	91.00	9,057.00	9,189.00	68.09	5.83	2.75
zziplib	5.10	1,158.00	1,206.33	0.44	3.54	0.31

TABLE 4: Average of Signature Generation & Signature Matching for each Program

4.6. Performance Comparison

Programs vs Signature. Table 4 shows the average size of signature, the time it takes to generate the signature, match the signature, and lastly, the time per version for each program in our dataset. As can be seen, the largest project we have in our project in Python and PHP, however, they do not affect the time of signature generation or matching. That is, although PHP is the largest project, it does not necessarily require a larger signature or a longer time to match. This is possible because we do not depend on the entire flow of a vulnerability that may travel across several files and operations of a larger project before coming to the manifestation point. Our process monitor helps us identify the files of importance and use that as a guide to create a signature.

Vulnerability vs Signature. On the other hand, Table 5 shows the relation between the vulnerability class and the size of the signature along with the time it takes to generate and match the signature. It shows that improper input validation (CWE-20) or numeric errors (CWE-189) have the largest signatures. Our manual analysis shows that these types of vulnerability can be complex in nature for a large project and take some time before they are triggered. Especially for a lot of these vulnerabilities are non-crashing making the signature even bigger. Additionally, a larger signature means more time to match the signature. Lastly, we see one particular class, use after free (CWE-416) takes the most time to be matched because of one particular CVE-2019-7317 which creates a large signature.

5. Discussion

5.1. Applied Analysis

To evaluate the real-world application of VERDIFF, we have incorporated it into the ongoing development and maintenance of `libtiff`. We have compiled all reported issues from April 15 – May 15, 2025, focusing on those deemed

CWE	Signature				Total
	#Nodes	#Edges	Gen.	Match	
CWE-119	497	525	0.51	2.45	0.11
CWE-125	1,609	1,682	0.38	5.06	0.42
CWE-189	16,759	16,866	1,465.01	141.22	30.97
CWE-190	6,866	7,044	21.68	1.42	1.78
CWE-20	55,115	57,950	665.34	64.04	56.24
CWE-416	7,602	7,792	21.10	808.64	23.05
CWE-476	985	1,036	0.51	4.96	0.20
CWE-787	452	471	0.08	2.21	0.08
Noinfo/other	1,294	1,317	0.26	5.35	0.33

TABLE 5: Average of signature generation & signature matching time for each vulnerability class

relevant to security and recognized by the developers. For example, developers ignore some vulnerabilities that result from improper use of internal functions. Our findings indicate that 95% of the reported security issues include PoC and ASAN reports derived from PoC. In total, we have identified four security vulnerabilities. These issues are reported to affect the latest version of `libtiff`, affecting any program that relies on this widely used library. At the time of writing, the developers have only resolved and closed 2 of these 4 issues. Moreover, although developers recognize these issues as security threats, none of them have been assigned a CVE.

VERDIFF analyzed 19 versions of `libtiff` (v4.0.0 to v4.7.0) for the issues analyzed, covering 13 years of development. Issues 684 (heap overflow) and 676 (memory leak) were found to affect all versions, while issue 682 (heap overflow) was only present in the latest. Notably, issues 684 and 676 require ASAN instrumentation to be detected, as they do not produce immediate crashes. Furthermore, older versions (before v4.2.0) use `automake` instead of `CMake`, complicating the compilation necessary for automated regression testing with PoC. The last one, issue 704 caused a crash in TaintGrind and thus could not be analyzed by VERDIFF, which is a rare and unrelated case that we have not seen before in our evaluation.

In general, the current reporting system only considers the latest version, leaving users unaware of the others. Incorporating VERDIFF and having users submit data flow along with already present PoC and ASAN reports allows a complete list of vulnerable versions to be compiled for external use in applying safety measures.

5.2. Reasoning Behind Inconsistency

We conducted a manual analysis of cases with high misclassification in the official report by examining their update history, initial issues, and vulnerability types. Additionally, we reviewed the CVE creation process. We find that the inconsistency in official vulnerability reports arises from a lack of comprehensive data collection, often relying solely on the descriptions of reporters. For example, the description of CVE-2017-16353 is as follows: "GraphicsMagick 1.3.26 is vulnerable to a memory information disclosure vulnerability found in the DescribeImage function of the magick/describe.c file, because of a heap-based buffer overread. " [20]. Based on this description, only a single version was marked vulnerable, overlooking 19 previous versions that were also susceptible. The process for listing affected versions hinges on three key phrases: i) in version (e.g., CVE-2017-16353) ii) before a particular version (e.g., in libpng 1.2.5 and earlier for CVE-2004-0597 [45]) iii) particularly mentioned versions (e.g., Python 3.x through 3.9.1 for CVE-2021-3177 [46]). In fact, one of the resources on how to report a vulnerability mentions these phrases as the right way to report a vulnerability [18]. Apart from the last description, the other two ways suffer from being incorrect or mislabeling versions. Thus, in addition to incorporating systems like VERDIFF the authorities also need to standardize how vulnerability descriptions are collected or what information is expected from the reporter.

5.3. Limitation

VERDIFF relies on the process monitor tool and data flow analysis report for dynamic properties to slice the fingerprinted AST. Data flow analysis has its own limitations affecting VERDIFF. Currently, partial signature matching is not supported. VERDIFF employs static analysis for vulnerability detection, facing its limitation as well. The limitations of VERDIFF are categorized into two sectors:

Limitation of Signature Matching. Currently, our signature does not support a partial matching, it is a hit-or-miss. However, better accuracy can be achieved by considering the importance of the node type that focuses more on control flow nodes, or such. On the other hand, such an approach also has to find a global threshold that marks the minimum match to call a version vulnerable.

Limitation of Static Analysis. One limitation that cannot be bypassed is the limitation of static analysis. We have already established the need for a static analysis in our case and we have tried to infuse dynamic properties in our signature. However, in the end, there could be changes in signature that do not necessarily change the execution of the program. However, our evaluation shows that VERDIFF can handle a variety of vulnerabilities. This gives us confidence that VERDIFF can provide an acceptable level of precision even with this limitation.

5.4. Future Work

We propose future works based on the limitation we have mentioned and some other places for general improvement.

Partial Signature Matching. To be able to match signature partially, we need a scoring system for the files and the nodes of the graph as well that will accurately distinguish between an important change and a non-important one. Furthermore, we also need to find a global threshold and verify that it holds for different kinds of program and vulnerability. Although challenging, if properly implemented, it can improve the accuracy of VERDIFF.

Root Cause Analysis. One of the limitations as described in the last section is the limitation of data flow analysis, and an automated *root cause analysis* will help to produce a more precise signature. However, it comes with its own challenges that need to be addressed. One of our future works is to use such an approach to calculate the overhead and effectiveness of using root cause trace instead of data flow.

6. Related Work

Related work in this sector can be divided into five categories, each addressing a different but related problem. Our framework tackles a novel problem, but its components are informed by the limitations of prior approaches:

- **Patch Detection:** Patch-based methods [61], [62], [64] use patches to locate vulnerabilities, creating signatures based on changes. These approaches require human input and timely patches, which are often delayed or ambiguous. In contrast, our problem setting cannot rely on patch availability or accuracy.
- **Clone Detection:** Works like [17], [23], [31] detect code clones via syntactic or semantic similarity. While [23] attempts functional similarity using deep learning, such methods need extensive training data. These approaches find full-function clones, whereas we seek partial similarity across versions, a different problem.
- **Vulnerability Transferability:** Research on PoC or exploit migration across versions [19], [29] addresses vulnerability propagation but is slow (e.g., VulScope [19] takes up to 8 hours/version) and not tailored for our real-time use case.
- **Vulnerability Detection:** Prior work focuses on specific vulnerability patterns (e.g., buffer overruns [56]) or uses ML with annotated datasets [38]. CPG-based approaches [26], [63], [66] are too resource-intensive for our lightweight execution-derived representation. Some tools like Arbiter [55], UAFChecker [67], IntTracer [15], and Tracer [32] are more aligned with our needs.
- **Vulnerable Code Reuse Detection:** Systems like MOVERY [60], V1SCAN [59], and V-SZZ [10] depend on patches or CVE data to track reused vulnerable code. Vuddy [35], later reused by V1Scan, generates signatures from patches. We use its signature matching mechanism to highlight its limitations in our setting.

Table 3 summarizes major related work, divided into two main categories. We select one representative from each for comparison with VERDIFF. Among publicly available patch-based tools, only Vuddy and V1Scan avoid compilation or expensive steps like fuzzing or ML; we choose Vuddy due to flaws in CVE-based data used by V1Scan. For the other category, Tracer is selected over Arbiter due to broader vulnerability coverage. While prior work focuses on detection, migration, or matching, VERDIFF offers a generalized and efficient solution for identifying vulnerable code and generating reusable signatures.

Vulnerable Code Detection	Work	Compilation	Fuzzing/ML	Granularity	Publicly Available?
Patch	**Vuddy [35]**	✗	✗	**Function**	✓
	V-SZZ [10]	✗	✓	Function	✓
	Movery [60]	✓	✗	Function	✓
	V1SCAN [59]	✗	✗	Function	✓
	BinXray [64]	✓	✗	Function	✗
	MVP [61]	✗	✗	Function	✗
	VIVA [62]	✓	✗	Function	✗
Predefined Signature	Arbiter [55]	✓	✗	Instruction	✓
	UAFChecker [67]	✗	✗	Instruction	✗
	IntTracer [15]	✓	✗	Instruction	✗
	VulSlicer [48]	✗	✗	Instruction	✗
	Tracer [32]	✓	✗	**Instruction**	✓

TABLE 6: Taxonomy of related work and their requirements
✓= requires ✗= does not require

7. Conclusion

In summary, the absence of detailed information regarding versions affected by a vulnerability endangers not just the program but also those who depend on it. Our analysis has highlighted the inconsistency of 27 CVEs that affect 11 well-known programs and libraries in real-world scenarios. We introduce VERDIFF, a system that can automatically assess all versions of a program for vulnerabilities using a payload-guided semantically enriched signature. Initially, we generate a vulnerability model by detecting the vulnerable code by observing the manifestation of the vulnerability given a specific triggering input. Next, we convert the source code of all versions into an abstract format and employ the model to derive a signature from the vulnerable abstract representation. Finally, we utilize this signature to identify the presence of the vulnerability in other versions of the program through subgraph isomorphism and established filtering techniques. Ultimately, VERDIFF allows us to effectively and efficiently detect 265 discrepancies in the reported versions.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive and insightful comments, which helped us improve the quality and clarity of this paper. This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contracts FA875024CB007, D24AP00313, and HR001125C0300. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not reflect the views of our sponsors.

References

[1] Greenbone openvas. Accessed on 2023-10-04. [Online]. Available: https://openvas.org/

[2] Hakiri. Accessed on 2023-10-04. [Online]. Available: https://hakiri.io/

[3] Oss index. Accessed on 2023-10-04. [Online]. Available: https://ossindex.sonatype.org/

[4] Tenable nessus. Accessed on 2023-10-04. [Online]. Available: https://www.tenable.com/products/nessus

[5] Veracode. Accessed on 2023-10-04. [Online]. Available: https://www.veracode.com/products/software-composition-analysis

[6] S. Ahmed, H. Liljestrand, H. Jamjoom, M. Hicks, N. Asokan, and D. D. Yao, "Not all data are created equal: Data and pointer prioritization for scalable protection against {Data-Oriented} attacks," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1433–1450.

[7] G. Audemard, C. Lecoutre, M. Samy-Modeliar, G. Goncalves, and D. Porumbel, "Scoring-based neighborhood dominance for the subgraph isomorphism problem," in *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings 20*. Springer, 2014, pp. 125–141.

[8] ——, "Scoring-based neighborhood dominance for the subgraph isomorphism problem," in *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings 20*. Springer, 2014, pp. 125–141.

[9] S. Axelsson, "The base-rate fallacy and the difficulty of intrusion detection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 3, pp. 186–205, 2000.

[10] L. Bao, X. Xia, A. E. Hassan, and X. Yang, "V-szz: automatic identification of version ranges affected by cve vulnerabilities," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2352–2364.

[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[12] E. Bendersky, "pycparser," 2023, [Online; accessed 29 June 2023]. [Online]. Available: https://github.com/eliben/pycparser

[13] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond heuristics: learning to classify vulnerabilities and predict exploits," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 105–114.

[14] ——, "Beyond heuristics: learning to classify vulnerabilities and predict exploits," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 105–114.

[15] X. Chen, "Inttracer: Sanitization-aware io2bo vulnerability detection across codebases," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 447–449.

[16] Z. Chen, X. Hu, X. Xia, Y. Gao, T. Xu, D. Lo, and X. Yang, "Exploiting library vulnerability via migration based automating test generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[17] M. Chilowicz, E. Duris, and G. Roussel, "Syntax tree fingerprinting for source code similarity detection," in *2009 IEEE 17th international conference on program comprehension*. IEEE, 2009, pp. 243–247.

[18] CVE, "Key details phrasing," https://www.cve.org/Resources/General/Key-Details-Phrasing.pdf, accessed: 2023-06-25.

[19] J. Dai, Y. Zhang, H. Xu, H. Lyu, Z. Wu, X. Xing, and M. Yang, "Facilitating vulnerability assessment through poc migration," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3300–3317.

[20] N. V. Database, "Cve-2017-16353 detail," November 2017, accessed on 2023-06-21. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-16353

[21] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *28th USENIX security symposium (USENIX Security 19)*, 2019, pp. 869–885.

[22] J.-D. Durand, "Marpax::languages::c::ast - translate a c source to an ast," 2023, [Online; accessed 29 June 2023]. [Online]. Available: https://people.eecs.berkeley.edu/~necula/cil/

[23] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 516–527. [Online]. Available: https://doi.org/10.1145/3395363.3397362

[24] K. A. Farris, A. Shah, G. Cybenko, R. Ganesan, and S. Jajodia, "Vulcon: A system for vulnerability prioritization, mitigation, and management," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 4, pp. 1–28, 2018.

[25] C. Fruhwirth and T. Mannisto, "Improving cvss-based vulnerability prioritization and response with context information," in *2009 3rd International symposium on empirical software engineering and measurement*. IEEE, 2009, pp. 535–544.

[26] Z. Guan, X. Wang, W. Xin, and J. Wang, "Code property graph-based vulnerability dataset generation for source code detection," in *Frontiers in Cyber Security: Third International Conference, FCS 2020, Tianjin, China, November 15–17, 2020, Proceedings*. Springer, 2020, pp. 584–591.

[27] G. Heon, "2023 unit 42 attack surface threat report." [Online]. Available: https://www.paloaltonetworks.com/resources/research/2023-unit-42-attack-surface-threat-report?utm_source=google-jg-amer-cortex&utm_medium=paid_search&utm_term=cyber+security+risk&utm_campaign=google-cortex-stsoc-amer-multi-lead_gen-en-eg-non_brand_broad&utm_content=gs-20340697332-156639569928-675898256381&sfdcid=7014u000001Rc86AAC&gad=1&gclid=Cj0KCQjwmvSoBhDOARIsAK6aV7ibPI7N9VLVel_P-2gQ5hGcHa89WrnudZSbHnX2U7xdgqBX4yCZiycaAhayEALw_wcB

[28] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 96–105.

[29] Z. Jiang, Y. Zhang, J. Xu, X. Sun, Z. Liu, and M. Yang, "Aem: Facilitating cross-version exploitability assessment of linux kernel vulnerabilities," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2122–2137.

[30] Z. Jiang, S. Gan, A. Herrera, F. Toffalini, L. Romerio, C. Tang, M. Egele, C. Zhang, and M. Payer, "Evocatio: Conjuring bug capabilities from a single poc," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1599–1613.

[31] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[32] W. Kang, B. Son, and K. Heo, "Tracer: signature-based static analysis for detecting recurring vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1695–1708.

[33] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, pp. 645–692, 2008.

[34] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005, pp. 187–196.

[35] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.

[36] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, pp. 384–417, 2018.

[37] S. Kwon, S. Woo, G. Seong, and H. Lee, "Octopocs: automatic verification of propagated vulnerable code using reformed proofs of concept," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 174–185.

[38] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[39] S. Magazine, "Security magazine: Average time to fix," 2023, [Online; accessed 12. May 2023]. [Online]. Available: https://www.securitymagazine.com/articles/95929-average-time-to-fix-severe-vulnerabilities-is-256-days

[40] C. McCreesh, P. Prosser, C. Solnon, and J. Trimble, "When subgraph isomorphism is really hard, and why this matters for graph databases," *Journal of Artificial Intelligence Research*, vol. 61, pp. 723–759, 2018.

[41] C. McCreesh, P. Prosser, and J. Trimble, "The glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants," in *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*, ser. Lecture Notes in Computer Science, F. Gadducci and T. Kehrer, Eds., vol. 12150. Springer, 2020, pp. 316–324. [Online]. Available: https://doi.org/10.1007/978-3-030-51372-6_19

[42] Mudongliang, "Linuxflaw: A repository of flaws in the linux kernel," 2018, accessed: 2024-08-26. [Online]. Available: https://github.com/mudongliang/LinuxFlaw

[43] G. Necula, "Cil - infrastructure for c program analysis and transformation," 2023, [Online; accessed 29 June 2023]. [Online]. Available: https://people.eecs.berkeley.edu/~necula/cil/

[44] V. H. Nguyen and F. Massacci, "The (un) reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013, pp. 493–498.

[45] NIST, "Nvd - cve-2004-0597," accessed on 2023-06-21. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2004-0597

[46] ——, "Nvd - cve-2021-3177," accessed on 2023-06-21. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-3177

[47] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1325–1341.

[48] S. Salimi and M. Kharrazi, "Vulslicer: Vulnerability detection through code slicing," *Journal of Systems and Software*, vol. 193, p. 111450, 2022.

[49] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[50] Snyk, "Synk: Open source security," 2023, [Online; accessed 12. May 2023]. [Online]. Available: https://snyk.io/reports/open-source-security/

[51] C. Solnon, "Alldifferent-based filtering for subgraph isomorphism," *Artificial Intelligence*, vol. 174, no. 12-13, pp. 850–864, 2010.

[52] O. Suciu, C. Nelson, Z. Lyu, T. Bao, and T. Dumitraș, "Expected exploitability: Predicting the development of functional vulnerability exploits," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 377–394.

[53] M. Tang, M. Alazab, and Y. Luo, "Big data for cybersecurity: Vulnerability disclosure trends and dependencies," *IEEE Transactions on Big Data*, vol. 5, no. 3, pp. 317–329, 2017.

[54] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards understanding third-party library dependency in c/c++ ecosystem," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3560432

[55] J. Vadayath, M. Eckert, K. Zeng, N. Weideman, G. P. Menon, Y. Fratantonio, D. Balzarotti, A. Doupé, T. Bao, R. Wang, C. Hauser, and Y. Shoshitaishvili, "Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 413–430. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/vadayath

[56] D. A. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities." in *NDSS*, vol. 20, no. 0, 2000, p. 0.

[57] B. Wang, K. Lu, Q. Wu, and A. Pakki, "Unleashing coveraged-based fuzzing through comprehensive, efficient, and faithful exploitable-bug exposing," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 5, pp. 2998–3010, 2021.

[58] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 1914–1927.

[59] S. Woo, E. Choi, H. Lee, and H. Oh, "{V1SCAN}: Discovering 1-day vulnerabilities in reused {C/C++} open-source software components using code classification techniques," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6541–6556.

[60] S. Woo, H. Hong, E. Choi, and H. Lee, "{MOVERY}: A precise approach for modified vulnerable code clone discovery from modified {Open-Source} software components," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3037–3053.

[61] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "Mvp: Detecting vulnerabilities using patch-enhanced vulnerability signatures." in *USENIX Security Symposium*, 2020, pp. 1165–1182.

[62] Y. Xiao, Z. Xu, W. Zhang, C. Yu, L. Liu, W. Zou, Z. Yuan, Y. Liu, A. Piao, and W. Huo, "Viva: Binary level vulnerability identification via partial signature," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 213–224.

[63] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, "Cpgva: code property graph based vulnerability analysis by deep learning," in *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*. IEEE, 2018, pp. 184–188.

[64] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 376–387.

[65] C. Yagemann, S. P. Chung, B. Saltaformaggio, and W. Lee, "Automated bug hunting with data-driven symbolic root cause analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 320–336.

[66] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[67] J. Ye, C. Zhang, and X. Han, "Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1529–1531.

[68] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2139–2154.

[69] S. Zampelli, Y. Deville, and C. Solnon, "Solving subgraph isomorphism problems with constraint programming," *Constraints*, vol. 15, pp. 327–353, 2010.

Appendix

1. Data Availability

The ethical considerations of our work will be addressed here. Our data collection methods were conducted exclusively through legitimate and public channels, such as NVD as provided by NIST and open source projects available on github, gitlab or sourceforge. The vulnerabilities analyzed were publicly disclosed and has been fixed in the latest release of these programs. We are in the process of communicating our finding with the authorities responsible to update the CVE reports.

2. Related Work Evaluation

In this section we detail the environmental setup we have done for each of our comparative work, Vuddy and Tracer.

2.1. Tracer. Tracer is built on top of Facebook's Infer, which necessitates the generation of a Makefile for a project. However, many projects only create the Makefile using the configure command once all necessary dependencies and their specific versions are satisfied. Therefore, we opted to execute Tracer in Docker containers. Specifically, we collected a Docker container that included all the Tracer artifacts and then employed our setup script to establish an environment for the target program. Afterward, we ran the configure command to produce the Makefile. Finally, we execute the scripts within the container and collect the results for the target version. Although infer has support for cmake, we could not make those work in our evaluation.

2.2. Vuddy. Vuddy processes the source code directory of a project to create a hidx file, which can then be uploaded to their website. This file includes signatures for all functions in each file within the project. Consequently, if any function corresponds to a vulnerability signature in their database, it can be identified. However, by default, it lacks the capability to focus on specific files or functions. We compiled the target files from all previous versions, placed them in a separate directory, and renamed them according to their original version. Additionally, we altered the Python signature generator script to record the function and file names along with the generated hash. With these changes, the hidx now includes hashes for all functions in our target files. We developed a script to extract and compare the hashes generated from the vulnerable version with all other versions.